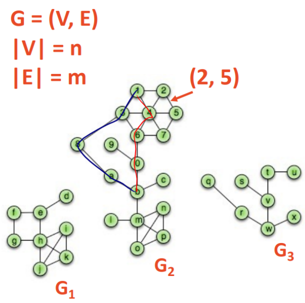
一些图的术语

$$G = (V, E)$$
$$|V| = n$$
$$|E| = m$$

Graph Vocabulary



- Incident edges:
  $\mathrm{I}(v) = \{(x, v) \in E\}$
- $\mathrm{Degree}(v) = |\mathrm{I}|$
- Adjacent Vertices:
  $\mathrm{A}(v) = \{x : (x, v) \in E\}$
- Path: 被边连起来的点的序列
- Cycle: 起点和重点相同的Path
- Simple Path: 没有自环和多重边（两个点之间有多条边）
- Subgraph:
  $G' = (V', E')$:
  $V' \in V, E' \in E$, and
  $(u, v) \in E' \implies u \in V', \quad v \in V'$

边数目的最值

下面$|V| = n, \quad |E| = m$

最小值：

- 不连通： 0
- 连通： $n-1$

最大值：

- 简单图： $\frac{n(n-1)}{2}$
- 非简单图： $+\infty$

Graph ADT

数据：

- 顶点
- 边
- 某种维护顶点和边关系的数据结构

方法：

- `insertVertex(K key);`
- `insertEdge(Vertex v1, Vertex v2);`
- `removeVertex(Vertex v);`
- `removeEdge(Vertex v1, Vertex v2);`
- `incidentEdges(Vertex v);`
- `areAdjecent(Vertex v1, Vertex v2);`

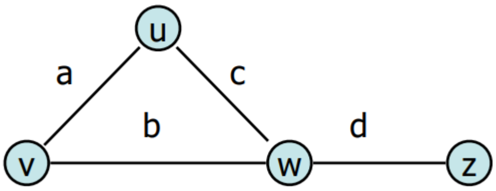
Directed graph only:

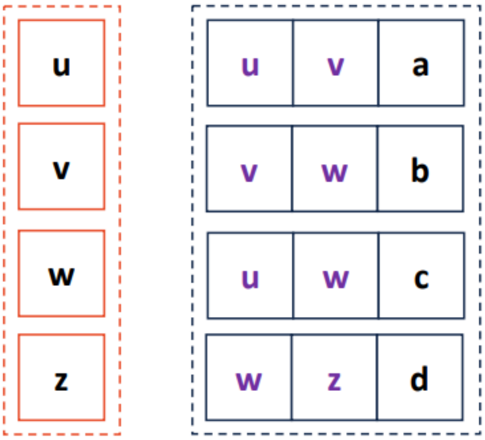
- `origin(Edge e);`
- `destination(Edge e);`

实现

表示一个图需要两个集合：顶点集和边集

Edge List

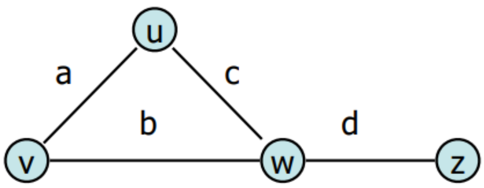


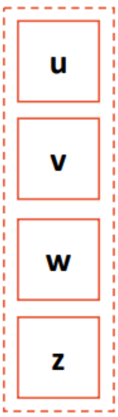


方法:

- `insertVertex(K key);` : $O(1)$
- `removeVertex(Vertex v);` : $O(m)$?
- `areAdjacent(Vertex v1, Vertex v2);` : $O(m)$
- `incidentEdges(Vertex v);` : $O(m)$

Adjacency Matrix 邻接矩阵



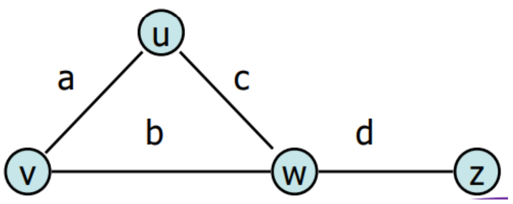


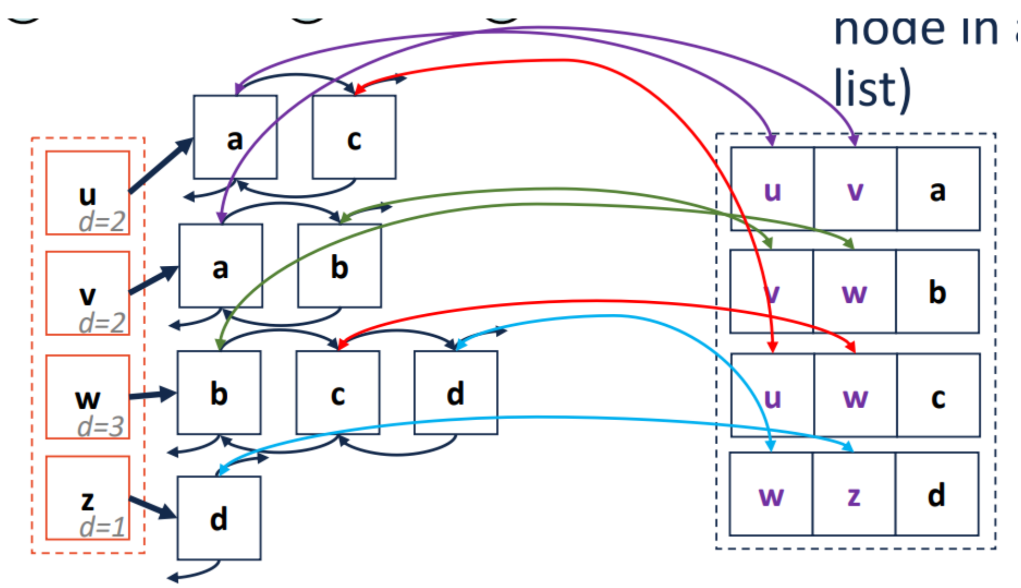
	u	v	w	z
u	0	1	1	0
v	1	0	1	0
w	1	1	0	1
z	0	0	1	0

方法（这里他课件写的贼乱，到时候重新看一下）：

- `insertVertex(K key);` : $O(n)$，因为要在顶点集插入点$O(1)$ + 在邻接表插入一行和一列$O(n)$
- `removeVertex(Vertex v);` : $O(n)$?
- `areAdjacent(Vertex v1, Vertex v2);` : $O(1)$
- `incidentEdges(Vertex v);` : $O(n)$

Adjacency List





同时维护一个edge list的时候是为了移除边的时候能更快

方法:

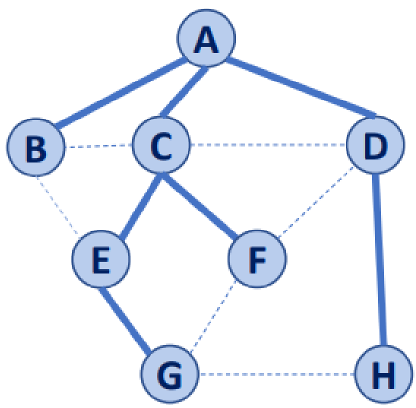
- `insertVertex(K key);` : $O(1)$
- `removeVertex(Vertex v);` : $O(\deg(v))$?
- `areAdjacent(Vertex v1, Vertex v2);` : $O(\min(\deg(v_1), \deg(v_2)))$
- `incidentEdges(Vertex v);` : $O(\deg(v))$

时间复杂度

Expressed as big-O	Edge List	Adjacency Matrix	Adjacency List
Space	n+m	n^2	n+m
insertVertex(v)	1	n	1
removeVertex(v)	m $\geq$ deg(v)	n $\geq$ deg(v)	deg(v)
insertEdge(v, w, k)	1	1	1
removeEdge(v, w)	1	1	1
incidentEdges(v)	m	n	deg(v)
areAdjacent(v, w)	m	1	min(deg(v), deg(w))

Traversal 遍历

BFS 宽度优先搜索



上图从A开始遍历

实线是discovery edge

虚线是cross edge

伪代码

```
1  BFS(G): # Init
2    Input: Graph, G
3    Output: A labeling of the edges on G as discovery and cross edes
4
5    foreach (Vertex v: G.vertices()):
6      setLabel(v, UNEXPLORED)
7    foreach (Edge e : G.edges()):
8      setLabel(e, UNEXPLORED)
9    foreach (Vertex v : G.vertices()): # 确保BFS能访问到每个元素
10     if getLabel(v) == UNEXPLORED:
11       BFS(G, v)
```

```
1  BFS(G, v):
2    Queue q
3    setLabel(v, VISITED)
4    q.enqueue(v)
5
6    while !q.empty():
7      v = q.dequeue()
8      foreach (Vertex w : G.adjacent(v)):
9        if getLabel(w) == UNEXPLORED:
10         setLabel(v, w, DISCOVERY)
11         setLabel(w, VISITED)
12         q.enqueue(w)
13       elseif getLabel(v, w) == UNEXPLORED:
14         setLabel(v, w, CROSS)
```

分析

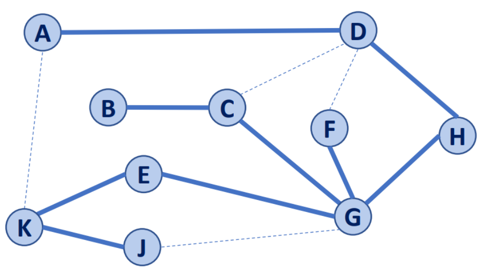
1. 由于上面第一个代码块的9-11行，可以正确处理disjoint graphs。如果要数有几个component，11行插入计数即可
2. 由于上面第二个代码块的第13-14行，我们可以检测环

3. 理想状态下运行时间为$O(n+m)$：我们只考虑第二个代码块，while循环运行n次，for循环运行$2m$次，因为$\sum \deg(v) = 2m$

观察

1. BFS可以找到a single source shortest path (SSSP)
2. cross edge和d (距离)的关系
   $|\mathrm{d}(u) - \mathrm{d}(v)| \leq 1$
3. discovery edges构成的结构是Minimum spanning tree (MST)
   好像是最小生成树
4. 遍历可以用来数components
5. 遍历可以用来检测环
6. 在BFS中，可以得到到每个顶点的最短距离
7. 在BFS中，cross edge的距离d的差值小于等于1
   $|\mathrm{d}(u) - \mathrm{d}(v)| \leq 1$

DFS 深度优先搜索



上图从A开始向右遍历

实线是discovery edge

虚线是back edge

```
DFS(G): # Init
  Input: Graph, G
  Output: A labeling of the edges on G as discovery and back edes
```

```

  foreach (Vertex v: G.vertices()):
    setLabel(v, UNEXPLORED)
  foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
  foreach (Vertex v : G.vertices()): # 确保BFS能访问到每个元素
    if getLabel(v) == UNEXPLORED:
      DFS(G, v)
```

```
DFS(G, v):
  setLabel(v, VISITED)

  foreach (Vertex w : G.adjacent(v)):
    if getLabel(w) == UNEXPLORED:
      setLabel(v, w, DISCOVERY)
      DFS(G, w)
    elseif getLabel(v, w) == UNEXPLORED:
      setLabel(v, w, BACK)
```

上面的第二个代码块还有一种实现方式

```
DFS(G, v):
  Stack s
  setLabel(v, VISITED)
  s.push(v)

  while !s.empty():
    v = s.pop()
    foreach (Vertex w : G.adjacent(v)):
      if getLabel(w) == UNEXPLORED:
        setLabel(v, w, DISCOVERY)
        setLabel(w, VISITED)
        q.push(w)
```

```
        elseif getLabel(v, w) == UNEXPLORED:
          setLabel(v, w, BACK)
```

可以看到这种方式仅仅是吧BFS的队列换成了栈